Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Workflow

Sébastien Mosser, Mireille Blay-Fornarino, Johan Montagnat

Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Workflow

Sébastien Mosser[⋆,†], Mireille Blay–Fornarino[⋆,†], Johan Montagnat[†]
⋆ University of Nice – Sophia Antipolis,
† CNRS, I3S Laboratory, MODALIS team,
Sophia Antipolis, France
{mosser,blay,johan}@i3s.unice.fr

Abstract

*Web Services Oriented Architecture (*WSOA*) supports development of high quality applications based on a control–flow between services. The introduction of a loop to deal with a dataset inside an orchestration is a typical evolution use case inside a* WSOA*, but there is no tool support to perform such refactoring operation. In this paper we propose a new method to refactor an orchestration dealing with a variable v so that it iterates over a dataset $v^\star$ automatically. This algorithm is then validated on a running software used as a validation platform by the French research project called* FAROS*.*

1. Introduction

Web Services Oriented Architectures provide a way to implement scalable Services Oriented Architectures (SOA, [4]) using web services as elementary services, and orchestrations [11] as composition mechanisms. The W3C defines orchestrations as "*the pattern of interactions that a Web Service agent must follow in order to achieve its goal*" [13]. Specialized (*i.e. elementary*) code is written inside web services, and each business process is described as an orchestration of those web services, defining a workflow using the BPEL language for example.

Evolution techniques like code refactoring [2] handle the redesign of a legacy application, following some well-known code rewriting techniques (interface extraction, inheritance, ...). These techniques are inspired by object–oriented concepts and typical object–oriented evolution use cases. Inside WSOA workflows, a typical evolution is to transform a data into a dataset and iterate computations over this dataset, *e.g.* for benchmarking purposes [8]. To the best of our knowledge, no existing refactoring technique is able to automatically transform an orchestration working on a single data v into an orchestration able to handle a dataset $v^\star \equiv \{v_1, \ldots, v_n\}$ (we use the * notation to denote a dataset).

The contribution of this paper is to define an algorithm able to automatically refactor an existing BPEL orchestration in this case. We identify in section 2 a typical example of loop introduction inside a legacy WSOA system named SEDUITE. In section 3, we enhance an evolution dedicated metamodel (ADORE) to handle this kind of evolution. Section 4 describes the evolution algorithm applied to a running example, and section 5 validates the algorithm inside a legacy platform. Section 6 presents relevant work in this field, and section 7 concludes this paper by showing some perspectives of this work.

2. Motivations: SEDUITE System & Data Sets

SEDUITE is an information system designed to fit academic institution needs. This system is used as a validation platform by the FAROS project[1]. SEDUITE is deployed inside an engineering school (POLYTECH'SOPHIA) and a specialized school for impaired children (IES CLÉMENT ADER). SEDUITE supports information broadcasting from academic partners (*e.g.* transport network, school restaurant) to several devices (*e.g.* user's smart-phone, PDA, desktop, public screen).

The system is built upon a WSOA. Sources of information such as timetables, bus schedules or weather forecasts are implemented as web services. Complex business processes used to combine theses data sources are defined as orchestrations, using the BPEL industrial standard [9]. Such an implementation follows WSOA methodological guidelines [10], positioning experimentations as a typical usage of a WSOA. More information about the system can be found on the project website[2].

[1] http://www.lifl.fr/faros
[2] http://anubis.polytech.unice.fr/jSeduite

A SEDUITE business process called `InfoProvider` defining an information retrieval process is described in FIG 1. A box represents an activity inside the process, and an arrow between two boxes means that the targeted box of the arrow is allowed to start at the end of the source one.

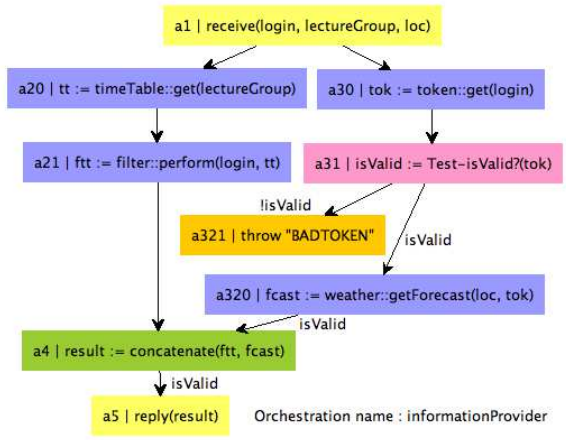

Figure 1. `InfoProvider` **orchestration**

When the `InfoProvider` orchestration receives an invocation message (a_1), it retrieves schedule information (a_{20}) and filters such schedule taking into account the login of the current user (a_{21}). Concurrently, it asks for an authentication ticket (a_{30}) and performs a validity test on the ticket (a_{31}). If the validity cannot be confirmed, the process throws an exception (a_{321}). In case of confirmation, it retrieves weather forecast information (a_{320}). A synchronization point appears when the orchestration concatenates filtered schedule and weather forecast information (a_4) before replying this list of information (a_5). This example is a simplified version of an existing orchestration which includes different sources and exception handling mechanisms.

The SEDUITE infrastructure is a perpetually evolving system, as new information sources are constantly developed and published. A typical kind of evolution is based on dataflow concepts. Originally, the `InfoProvider` process was defined to handle a single *lectureGroup* input parameter. But when new educational programs are proposed inside the school building, the orchestration has to be refactored to handle such a dataset containing all the existing programs. It results in a huge and deep modification of the original process, introducing a loop over the impacted activities, adding some new activities to read the current program from the set, and then appending the result to others, ...

To perform such a modification, the administrator has to clearly identify impacted activities, introduce a loop activity around them, define new variables and new assignments. This kind of situation is a typical evolution inside SEDUITE orchestrations (handling a set of lecture groups, a set of bus lines, a set of classes ...). But there is no support from existing tools to automate the computation of the resulting orchestration. Consequently the administrator always performs the modification process by hand. Such a method is error prone and time consuming.

3. Adding Datasets Concerns Inside ADORE

The ADORE metamodel reifies orchestrations behaviours. It is designed to focus on orchestrations evolution, proposing an algorithm to automatically support behavioral evolution of orchestrations through a formal merge process. The metamodel and the proposed behavioral merge process are fully described in [6]. We propose to enhance ADORE to introduce dataflow changes inside its evolution capabilities.

As a running example, we define a simple $adder$ orchestration illustrated in FIG. 2. This orchestration receives an invocation message containing two input string parameters x and y (a_1). It retrieves from a *memory* web service (a_{20}, a_{21}) the integer value associated to x in a variable x_{int} (resp. y_{int}) and returns (a_4) a result r (computed as $x_{int} + y_{int}$ in a_3).

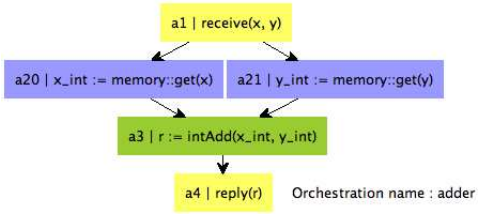

Figure 2. $adder$ **orchestration**

Our goal is to produce through our algorithm an orchestration $adder^{(x*)}$ able to handle a set of values $x^\star \equiv \{x_1, \ldots, x_n\}$ instead of x. As the original process computes its result r from x_{int} and y_{int}, we wants to compute a set of result $r^\star \equiv \{r_1, \ldots, r_n\}$, where $r_i = x_{int_i} + y_{int}$. Such an evolution implies *(i)* a modification of the orchestration business process itself (its behaviour) to handle the dataset and *(ii)* a modification of the orchestration external interface (its WSDL contract). In this paper we focus our contribution on the behaviour modification.

$$
\begin{aligned}
adder \quad &: \quad String \times String \rightarrow Integer \\
&\quad\quad x \times y \mapsto x_{int} + y_{int} \\
adder^{(x^\star)} \quad &: \quad String^\star \times String \rightarrow Integer^\star \\
&\quad\quad \{x_1, \ldots\} \times y \mapsto \{x_{int_1} + y_{int}, \ldots\}
\end{aligned}
$$

3.1. ADORE: An Activity Meta–model

Conforming to the ADORE metamodel, an orchestration is defined as a set of activities $\mathcal{A}^\star$ and a partial order $\prec^\star$ between those activities. An activity a is defined as a tuple identified by a unique Id. Each activity is characterized by its kind, and uses a list of variables as inputs. An activity can use a variable as its output. Activity kinds are defined as a subset of BPEL specifications. The different kinds defined inside the metamodel handle *(i)* web service invocation (`invoke`), *(ii)* variable manipulation (`assign`), *(iii)* fault report (`throw`), *(iv)* boolean predicate evaluation (`test`), *(v)* message reception (`receive`) and *(vi)* response sending (`reply`).

$$\begin{aligned} O &\equiv (\mathcal{A}^\star, \prec^\star) \\ \forall a \in \mathcal{A}^\star, a &= (Id, Kind, Inputs, Output) \\ (a_1 \prec a_2) \in \prec^\star &\equiv start(a_2) \Rightarrow end(a_1) \end{aligned}$$

The ADORE metamodel is designed to support orchestration evolutions. To fill the gap between ADORE and the industrial description as BPEL code, we develop a set of model transformations between the BPEL language and the ADORE metamodel. An orchestration defined as BPEL code can be transformed into its representation conforming to the ADORE metamodel. The reciprocal transformation generates BPEL entities from an ADORE representation. The `adder` orchestration described in FIG 2 corresponds to the following model, conform to ADORE metamodel.

$$\begin{aligned} adder &\equiv (\{a_1, a_{20}, a_{21}, a_3, a_4\}, \\ &\quad \{a_1 \prec a_{20}, a_1 \prec a_{21}, a_{20} \prec a_3, \ldots\}) \\ a_1 &\equiv (a_1, receive, \{x, y\}, \emptyset) \\ a_{20} &\equiv (a_{20}, invoke(memory, get), \{x\}, \{x_{int}\}) \\ a_{21} &\equiv (a_{21}, invoke(memory, get), \{y\}, \{y_{int}\}) \\ a_3 &\equiv (a_3, assign(intAdd), \{x_{int}, y_{int}\}, \{r\}) \\ a_4 &\equiv (a_4, reply, \{r\}, \emptyset) \end{aligned}$$

3.2. Dataflow Concepts Inside ADORE

As ADORE is a control–flow driven evolution metamodel, we need to enhance the metamodel with new functions to handle datasets driven evolutions such as loops introduction.

DataFlow ($\mathcal{D}_\mathcal{F}$): Following the W3C definition, an orchestration represents a control–flow. Identifying data flows inside an orchestration allows us to extract activities inside the control flow which derive from the usage of a given variable. We define $\mathcal{D}_\mathcal{F}$ as a transitive closure of all activities *impacted* by the evolution of a variable v into a dataset $v^\star$: $\forall a_i \in \mathcal{D}_\mathcal{F}$, a_i either uses v as input variable or a_i use as input variable a variable transitively computed from v. As an example, the data flow extracted from the variable x inside the $adder$ orchestration is defined as the following: $\mathcal{D}_\mathcal{F}(\{a_1, a_{20}, a_{21}, a_3, a_4\}, x) = \{a_1, a_{20}, a_3, a_4\}$.

Firsts ($\mathcal{F}$) & Lasts ($\mathcal{L}$): We define two auxiliary functions to deal with set of activities, named $Firsts$ and $Lasts$. We called $Firsts$ (resp $Lasts$) of a set of activities all activities that have no predecessors (resp. successors) inside this set of activities.

For example, $\mathcal{F}(\prec^\star, \mathcal{D}_\mathcal{F}(\{a_1, a_{20}, a_{21}, a_3, a_4\}, x)) = \{a_1\}$ and $\mathcal{L}(\prec^\star, \mathcal{D}_\mathcal{F}(\{a_1, a_{20}, a_{21}, a_3, a_4\}, x)) = \{a_4\}$.

Interface Activities: We call *interface activities* of a variable v inside a set of activities $\{a_1, \ldots, a_n\}$ all activities $\{a_i, \ldots, a_j\}$ that exchange (*ie* receive or send) v with external entities. These activities can be *(i)* a `receive` or a `reply`, *(ii)* a `throw` or *(iii)* an `invoke` using variable v.

For example, interface activities of variables x inside `adder` activities are defined as the following: $Interface(\{a_1, a_{20}, a_{21}, a_3, a_4\}, x) = \{a_1\}$

Core ($\mathcal{C}$): We define the *core* $\mathcal{C}(o, v)$ of a dataset evolution as a set of activities built upon a *dataflow*, without interface activities of *(i)* input variables of $Firsts$ activities and *(ii)* output variables of $Lasts$ activities. It represents the set of activities $\{a_i, \ldots, a_j\}$ defined in an orchestration o that need to be included in a loop to support the $v \mapsto v^\star$ evolution.

As an example, the *core* activities set $\mathcal{C}(adder, x)$ consists in all activities inside $\mathcal{D}_\mathcal{F}(Activities(adder), x)$ (*ie* $\{a_1, a_{20}, a_3, a_4\}$), without its $firsts$ (respectively $lasts$) interface activities a_1 (resp. a_4). As a result, $\mathcal{C}(adder, x) = \{a_{20}, a_3\}$

4. Set–Evolution Algorithm: $v \mapsto v^\star$

The set–evolution algorithm runs over the previously defined *core* activities and computes from a given input variable v inside an orchestration o the set of *"actions"* to execute in order to enhance o to handle $v^\star$. Atomic actions available following the ADORE metamodel are entities creation or deletion (*addOrder, addActivity*, *delOrder*, ...) or symbolic variable substitution σ as defined by Stickel in [12]. After executing these actions, o behaviour is able to handle a dataset $v^\star$ instead of the legacy v.

We illustrate the algorithm using the previously defined `adder` orchestration. Each step of the algorithm is described and then applied to the running example, *i.e.* automatically produce an orchestration $adder^{(x^\star)}$ which handles a dataset $x^\star \equiv \{x_1, \ldots, x_n\}$ instead of x.

4.1. Generic Algorithm: $SetEnhance(o, v)$

The algorithm results in the introduction of a loop activity in the BPEL orchestration to iterate over all values in $v^\star$. It is a four step process: firstly, it extracts the evolution core $\mathcal{C}(o, v)$, which represent the body of the computed loop; secondly a variable substitution $v \mapsto v^\star$ is performed outside the *core* activities; thirdly the algorithm introduces special activities inside the *core* to interface the loop with the rest of the orchestration; and finally the order relation inside the orchestration is rearranged to take care of the loop introduction. FIG 3 illustrates the resulting workflow in the $adder$ example case.

1. Core extraction: Using previously defined functions and properties, the core $\mathcal{C}(o, v)$ is computed. It represents the body of the computed loop we are going to build. In our example, $\mathcal{C}(adder, x) = \{a_{20}, a_3\}$. These two activities correspond to the body of the loop that processes the set of input data $\{x_1, \ldots, x_n\}$.

2. Variable substitution: In a generic way, the algorithm builds a loop able to handle a dataset as input, and return a dataset as output. Outside the *core*, these variables must be renamed (using a substitution $\sigma(v \rightarrow v^\star)$) to match with the new name.

In the `adder` example, we have to perform two substitutions: the input parameter x must be renamed as $x^\star$, and the output variable r must be renamed as $r^\star$, as it will correspond to a set of results $\{r_1, \ldots, r_n\}$. Those substitutions $\sigma(x \rightarrow x^\star)$ and $\sigma(r \rightarrow r^\star)$ must be applied on activities defined outside of the core (message reception as a_1 and response sending as a_4).

3. Loop Creation & Adaptation: A loop is defined as a composite activity, embedding a set of activities and an order relation between these activities. Contrarily to *simple* activities, an activity expressed inside a *loop* can be executed several times (one time per iteration, possibly concurrently). The computed loop is defined as a new activity inside the orchestration, which contains the core activity set and the existing order relation between these activities. Inside the loop, two special assignment activities ($feeder$ and $sweller$) are defined to handle the data–set semantic. The $feeder$ reads the current element of the dataset $v^\star$ and assigns it into v. The reciprocal activity $sweller$ adds the current result to the results set. The order relation is enriched to set up the $feeder$ as the first activity inside the loop, and the $sweller$ as the last one.

In the `adder` example, the $feeder$ is defined as an assignment from $x^\star$ into x using the $feed$ assignment function. In a reciprocal way, the $sweller$ is defined as an assignment from r into $r^\star$ using the $swell$ function (which basically concatenate the current result with the previously computed results set). The loop is defined as a new activity l_1, which contains the activity set $\{l_1\_feeder, a_{20}, a_3, l_1\_sweller\}$. The process defines two new orders entity inside the order relation to ensure the $feeder$ is the first activity and the $sweller$ is the last one: $l_1\_feeder \prec a_{20}$ and $a_3 \prec l_1\_sweller$.

4. External Activities Reordering: The order relation $\prec^\star$ defined inside o is inconsistent for now. An order $ext \prec body$ (respectively $body \prec ext$) between an activity ext defined outside the loop and $body$ defined inside the loop does not make any sense in front of BPEL engines capabilities that do not handle such links. Those orders entities must be detected and then reordered to target the loop instead of the $body$ activity. The same kind of reciprocal erroneous orders entities (an activity inside the loop targeting an activity outside the loop) are reordered to start after the $sweller$. Patterns like $b_1 \prec ext \prec b_2$ which produce deadlock order relations after substitution (the loop containing b_1 and b_2 can start at the end of ext, but ext needs to start after the end of the loop ...) are detected using graph–pattern recognition techniques and highlighted for the user. She can decide to introduce est inside the loop, or choose to execute it $before$ or $after$ the loop.

In our running example, three order entities between the outside and the body of the loop exist inside the `adder` example: $a_1 \prec a_{20}$, $a_{21} \prec a_3$ and $a_3 \prec a_4$. These entities must be deleted and replaced by proper entities where each activity inside the loop is replaced by the loop identifier. The correct order entities for this special example is defined as the following: $a_1 \prec l_1$, $a_{21} \prec l_1$, $l_1 \prec a_4$. The resulting orchestration `adder`* is described in Fig 3.

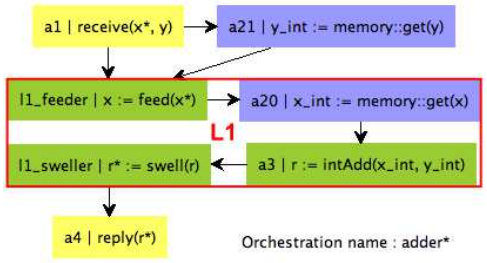

Figure 3. $adder^{(x^\star)} \equiv setEnhance(adder, x)$

4.2. Specialization Using Loop "Make–Up"

The algorithm defined in the previous section is a generic algorithm which automatically formats the output of the

loop as a list corresponding to a concatenation of the results dataset. In some cases, this generation is too basic and it does not reflect the expected semantics of the set–evolution. For instance when working on numeric data, a common need is to compute a global statistical result such as the average of a set of results rather than handling individual values.

To handle this diversity of possible algorithm specialization, we define a set of "cosmetic" activities, like flattening a list or computing an average. These activities are defined in an evolution toolbox and come with a set of well-known properties like associativity or commutativity to encourage further composition. As we identify during the loop creation special key points ($feeder$, $sweller$, ...), the user can manually insert a cosmetic activity to a key point to perform final optimization. ADORE conflicts detections rules set can detect inconsistencies introduced by the make–up step (*e.g.* using a dataset as input parameter instead of a scalar), but it is under user's responsibility to solve those conflicts using her knowledge of the application.

4.3. Implementation

The evolution algorithm is implemented using the PROLOG language, as it intensively uses inference rules. The algorithm is defined as a rule `setEnhance(+O,+V,-ToDo)`. From an orchestration O and a variable V, it unifies with $ToDo$ the resulting list of atomic actions to perform on O to handle the set enhancement of V. These actions are defined inside the ADORE metamodel and available from the project website[3].

5. Validation: Handling Sets Inside SEDUITE

Coming back to our first example, we use the SEDUITE infrastructure deployed inside the FAROS French national research project to validate the algorithm. We define orchestrations using scalar variables and then compare the new orchestrations obtained using our algorithm with legacy codes deployed on SEDUITE servers. In this paper, we focus on the simplified `InfoProvider` described in section 2 for a better understanding.

Generic usage: handling $lectureGroup^{\star}$. In this section, we apply the algorithm to `InfoProvider` to handle as input variable a set of groups $lectureGroup^{\star}$ instead of a single one. This situation happens in the POLYTECH'SOPHIA system when new educational programs were proposed inside the school. The public displays process which broadcasts timetables must be adapted to broadcast the set of timetables information instead of a single one.

[3] `http://rainbow.i3s.unice.fr/adore`

To perform the action, we first load inside the logical engine `seduite` (the orchestration conforming ADORE meta–model) and `enhance` (the evolution algorithm) source files. We apply the `setEnhance` rule to `informationProvider` orchestration and `lectureGroup` variable. The engine unifies `ToDo` with a set of `Adore` actions, as shown in the following listing.

```
?- [seduite,enhance].
?- setEnhance(informationProvider,
                lectureGroup,ToDo).
ToDo = [addActivity(l1,loop(
 [l1_feeder,a20,a21,a4,l1_sweller]),[],
 []), delOrder(a1,a20), addOrder(a1,l1),
 sigma(..., ...)|...]
```

The generic algorithm computes a loop l_1 built around three activities: schedule information retrieval (a_{20}), filtering (a_{21}) and finally the concatenation of schedule information with weather forecast information (a_4). All activities that do not interfere with $lectureGroup$ dataflow stay intact inside the resulting orchestration (*ie* $\{a_{30}, a_{31}, a_{321}, a_{320}\}$). The interface activities (message reception a_1 and response sending a_5) are not included inside the loop.

Loop "make-up": `eject` & `flatten`. Without any additional information, the algorithm will produce a cartesian product of each schedule $\{ftt_1, \ldots, ftt_n\}$ with the forecast information $fcast$. The result variable cross each ftt_x with the forecast information, and return a list of tuples $\{\{ftt_1, fcast\}, \ldots, \{ftt_n, fcast\}\}$. In SEDUITE semantic, this is not the expected behaviour: the administrator wants to add all schedules into the information list instead of producing a list of information tuples.

`eject`: To produce this kind of behaviour, the administrator just has to `eject` the concatenation activity a_4 of the loop. With this new information, the algorithm stop the loop body around a_{20} and a_{21}. With this body, the $sweller$ activity will naturally add the computed list of schedule information into the results list: $\{\{ftt_1, \ldots, ftt_n\}, fcast\}$.

`flatten`: To respect the external interface of the `InfoProvider` orchestration, the administrator adds a `flatten` assignment before the response sending a_5. With this new activity, the result list conforms to the external interface and with the orchestration semantic: $\{ftt_1, \ldots, ftt_n, fcast\}$

6. Related Work

Array programming [3] relies on efficient access and operation on arrays data. This paradigm defines several operators which work equally on arrays and on scalar data.

Applications are then defined as the composition of operators. Current approaches try to introduce array programming concepts inside others existing paradigms like object–oriented languages for example [7]. Contrarily to this method, we propose to introduce the set concept at the model level to be able to reason on it, and then translate this concept into existing concepts (loop) in the targeted technology.

The grid–computing research field deals with large–scale data driven workflows. Grid workflows are data–intensive so there is a need for grid users to handle data-set composition at a high level of abstraction. Data composition operators were defined in [5] allowing composition through iteration strategy (one to one, one to many). These operators ($\otimes$, $\odot$) can be naturally composed. Such operator composition can help the loop introduction techniques when trying to introduce an iteration around several variables instead of a single one.

Mashups–driven application came from Web 2.0 movement. Mashups allow *"end–user programming of the Web"* [14], making web surfer able to compose data sources in a friendly way. Mashups invade scientific field of research like bioinformatic [1]. As mashups designers are not computer sciences specialists by essence, automatic tools dealing with iteration can ensure the correctness of builded flows.

7. Conclusions & Perspectives

In this article, we address the problem of loops introduction inside existing orchestrations from WSOA systems. We motivate this work using as a running example a legacy application called SEDUITE. This system is used as a validation platform by the FAROS French research project. We propose to implement the loop introduction technique inside a formal meta–model called ADORE, as it is dedicated to WSOA evolution. The technique is applied inside the existing SEDUITE platform deployed in an academic institution.

The algorithm presented here sketches the loop introduction techniques inside ADORE. Currently, the algorithm produces a BPEL orchestration that cannot be modeled in ADORE and consequently cannot be reprocessed in this framework (due to the loop activity that is not part of ADORE's metamodel). A perspective is to produce an ADORE-compliant orchestration as output.

Grid computing workflows handle large datasets. Dealing with a dataset at the design level is an error–prone and time consuming process. Using techniques drifting from the loop introduction presented here, a designer will be able to design her flow reasoning on a single data and then enhance the flow to automatically handle a dataset.

Acknowledgments

This project is partially funded by the French Research Agency (ANR) through the FAROS project. The ADORE framework is one of the platforms targeted by FAROS. The SEDUITE software and its reference implementation is used as a validation platform.

References

[1] F. Belleau, M.-A. A. Nolin, N. Tourigny, P. Rigault, and J. Morissette. Bio2rdf: Towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, March 2008.

[2] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[3] H. Hellerman. Experimental personalized array translator system. *Commun. ACM*, 7(7):433–438, 1964.

[4] M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz. Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, Feb. 2006.

[5] J. Montagnat, T. Glatard, and D. Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France, June 2006.

[6] S. Mosser, M. Blay-Fornarino, and M. Riveill. Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution. In *2nd European Conference on Software Architecture (ECSA'08)*, Paphos, Cyprus, Sept. 2008. Springer LNCS.

[7] P. Mougin and S. Ducasse. Oopal: integrating array programming in object-oriented programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 65–77, New York, NY, USA, 2003. ACM Press.

[8] C. Nemo, T. Glatard, M. Blay-Fornarino, and J. Montagnat. Merging overlapping orchestrations: an application to the Bronze Standard medical application. In *International Conference on Services Computing (SCC 2007) AR = 20%*, pages 364–371, Salt Lake City, Utah, USA, July 2007. IEEE Computer Engineering.

[9] OASIS. Web services business process execution language version 2.0. Technical report, OASIS, 2007.

[10] M. P. Papazoglou and W. J. V. D. Heuvel. Service oriented design and development methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442, 2006.

[11] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[12] M. E. Stickel. A unification algorithm for associative-commutative functions. *J. ACM*, 28(3):423–434, 1981.

[13] W3C. Web service glossary. Technical report, 2004.

[14] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444, New York, NY, USA, 2007. ACM Press.